4存储器、连续分配

2018年11月3日 8:26

-
- 存储器的层次结构

1. 存储器仍是宝贵而稀缺的资源
2. 外存的文件管理与内存的管理类似，将在第7章介绍

一. 多层结构的存储器系统

1. 多层结构：最高层CPU寄存器、中间主存、最底层辅存

1)

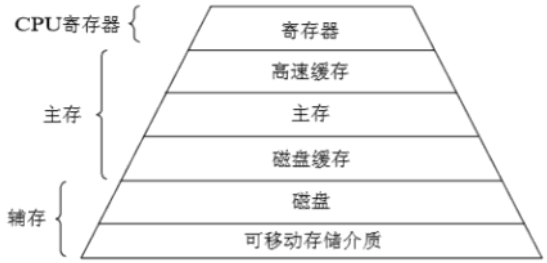


图 4-1 计算机系统存储层次示意

2) 层次越高越靠近CPU，越需要速度快，价格也越高

3) 存储器速度必须非常快才能不影响处理机的运行速度（同理I/O速度也远低于内存），因而需要缓存

4) 寄存器、主存（高速缓存、主存储器、磁盘缓存）属于操作系统管辖范畴，掉电后信息不再存在；辅存则属于设备管辖范围、存储的信息将被长期保存

2. 可执行存储器：寄存器和主存储器

1) 用load/store指令即能访问可执行存储器、而访问辅存需要通过I/O设备实现
2) 因为不涉及中断、设备驱动、物理设备、所以访问时间一般少了3个数量级

二. 寄存器和主存储器

1. 主存储器：简称内存/主存

1) 用于保存进程运行时的程序和数据，CPU与外界的交流依托其地址空间
2) 早期磁芯内存只有几十几百K，现在VLSI内存一般微机至少数十M到数G，嵌入式也有几十K到几M

2. 寄存器：与处理机同速，完全能与CPU协调工作

1) 用于存放处理机运行时的数据（操作数、地址）以加速存储器访问速度
2) 现在一般微机有数十数百个字长32或64位的寄存器；嵌入式仍不超过十几个，常为8位

三. 高速缓存和磁盘缓存

1. 高速缓存：备份主存中常用数据，减少CPU对主存的访问，大幅提高执行速度

1) 容量远大于寄存器，比内存越小两三个数量级，即几十K到几M
2) 访问速度快于主存储器，许多地方都设置了高速缓存以缓和速度矛盾
3) 程序执行的**局部性**原理：较短时间内，程序执行仅局限于某个部分；访问过的数据在短时间内会再被访问
   (1) 快要执行某段程序时，检查确定它不在高速缓存中，就临时复制进去
   (2) 下一条待执行指令也应提前准备在指令高速缓存中
4) 越快越贵，一般分几级，紧靠内存的一级最快，容量最小

2. 磁盘缓存：暂存频繁使用的部分磁盘数据和信息，减少主存访问磁盘次数

1) **磁盘缓存一般是主存中划出的一部分**，而**高速缓存是独立硬件存储器**
2) 辅存数据必须先进入主存才能给CPU用，整个主存都勉强可视作辅存缓存

3) 有些系统自动把老文件数据从辅存转储到磁带等海量存储器上，降低存储价格

-
- 程序的装入和链接

1) 用户源程序执行前，需要步骤：

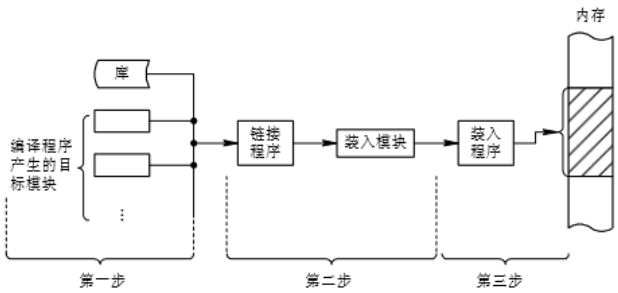
1. 由编译程序Compiler编译成若干目标模块ObjectModule
2. 由链接程序Linker将目标模块和库函数链接，形成完整装入模块LoadModule
3. 由装入程序Loader将装入模块装入内存
4. 

图 4-2 对用户程序的处理步骤

一. 程序的装入

1. 绝对装入方式(Absolute Loading Mode)
   1) 仅适用于单道系统，提前将绝对（物理）地址放入目标代码，装入程序按该地址装入
   2) 一般由编译或汇编时算出地址，也可由程序员算出并赋予。一般是编译或汇编时由符号地址转换的
2. 可重定位装入方式(Relocation Loading Mode)
   1) 多道程序不可能预知地址，只能根据内存情况装入合适位置
   2) 修改过程：将逻辑地址加上程序在内存中的首物理地址
   3) **重定位：装入时对指令和数据地址的修改过程**
   4) 静态重定位：在装入时连接装入程序一次完成重定位，以后不再改变
3. 动态运行时装入方式(Dynamic Run-time Loading)
   1) 运行过程中进程在内存中的位置可能经常要改变，使物理地址也改变
      (1) 如有对换功能的系统对一个进程多次换出换入
   2) 动态重定位：**推迟到程序执行时才重定位**，即装入内存后所有地址仍是逻辑地址
   3) 为使地址转换不影响指令执行速度，需要重定位寄存器等硬件的支持

二. 程序的链接

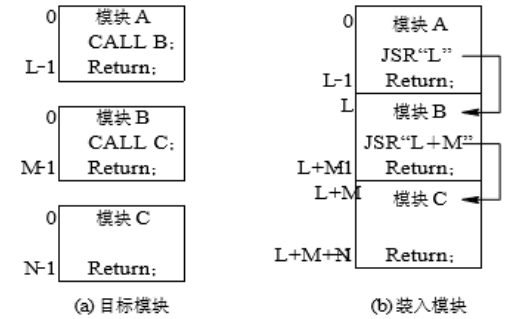
1. 静态链接方式(Static Linking)：运行前先把模块和库函数链接在一起，不再拆开
   1) 修改各模块的相对地址
   2) 变换各模块的外部调用符号为相对地址
   3) 

图 4-4 程序链接示意图
2. 装入时动态链接(Load-time Dynamic Linking)：**调用并装入模块时**修改相对地址
   1) 便于修改和更新、便于共享模块
3. 运行时动态链接(Run-time Dynamic Linking)：

1) 对某些模块的**链接推迟到程序执行时再由OS寻找并装入后**

2) 加快了程序装入过程，节省了大量内存空间

-
- 连续分配存储管理方式

一. 单一连续分配

1) 单道程序中，内存分成系统区和用户区两部分

(1) 系统区：仅供OS使用，通常放在内存低址部分

(2) 用户区：被一道程序独占

1. 存储器保护机构：防止用户程序对操作系统的破坏

1) 用户程序自己破坏操作系统后果并不严重，只是影响程序运行而已，操作系统可根据系统再启动而重新装入内存，因而当时有的电脑并没有这个机构

二. 固定分区分配

多道程序系统中，用户空间被划分为若干固定大小的区，每个分区装一道作业

1. 划分分区的方法：

1) 分区大小相等：不灵活，易浪费；控制相同对象（如炉温群控）时较方便

2) 分区大小不等：灵活，一般是较多小分区，少量大分区

2. 内存分配

1) 按分区大小建立分区使用表，包含地址、大小、分配状态

2) 分配程序按程序大小检索该表，将第一个满足的未分配区改成已分配

3) 由于大小固定，仍然可能造成浪费

4)

分区号	大小/KB	起址/KB	状态
1	12	20	已分配
2	32	32	已分配
3	64	64	已分配
4	128	128	未分配

(a) 分区说明表

(b) 存储空间分配情况

图 4-5 固定分区使用表

三. 动态分区分配/可变分区分配

1. 动态分区分配中的数据结构

1) 空闲分区表：每个空闲分区占一个表目，包括区号、大小、地址

2) 空闲分区链：一个双向链，分配后状态位由0改为1

3)

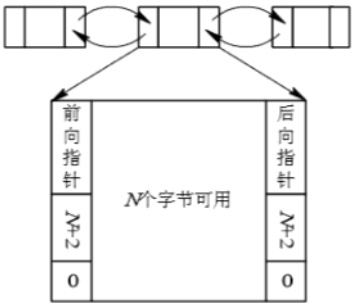


图 4-6 空闲链结构

2. 动态分区分配算法：对系统性能有很大影响。详见后几目

3. 分区分配操作：

1) 分配内存：先设置一个最小分区size

(1) 找到第一个大于内存请求的分区大小

(2) 计算分配后盈余是否大于size，是的话就只划分出一小块分区给它

(3) 然后把应分配的空间地址分给请求者，修改相关数据结构

(4)

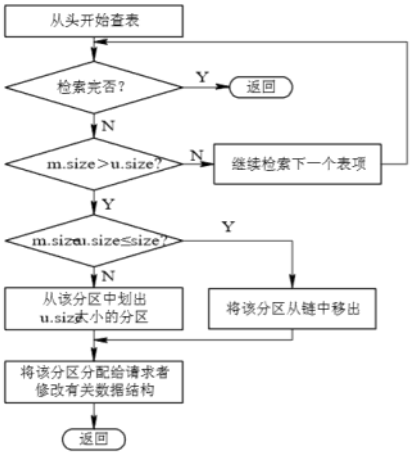


图 4-7 内存分配流程

2) 回收内存：进程运行完释放内存时，按地址找到该插入（链）表的地方

(1) 只和前一空闲分区相邻：增加前区大小

(2) 只和后一空闲分区相邻：修改后区地址为该区地址，增加后区大小

(3) 和前后空闲分区都相邻：取消后区，前区大小增加该区和后区

(4) 和前后空闲分区都不相邻：在该插的地方新建表项

四. 基于顺序搜索的动态分区分配算法

1. **首次适应**算法(first fit)：每次从链首顺序查找直到找到足够大的分区
   1) 优先利用低址分区，让后到的大作业有条件分配到大空间
   2) 低址分区被不断划分，留下很多难用的碎片，增加下次查找的开销
2. **循环首次适应**算法(next fit)：设置起始查寻指针指向上次分配区的下一个
   1) 空闲分区分布均匀，减少了查找开销
   2) 缺乏大空闲分区
3. **最佳适应**算法(best fit)：找满足要求的最小空闲分区
   1) 一般是按大小升序排列空闲分区链，也较快
   2) 容易留下难以利用的碎片
4. **最坏适应**算法(worst fit)：每次挑最大的空闲分区，割一部分下来
   1) 一般是按大小降序排列空闲分区链，缺乏大分区
   2) 出现碎片的可能性最小，对中小作业有利，查寻只需一次

五. 基于索引搜索的动态分区分配算法

1. **快速适应**算法(quick fit)/分类搜索法
   1) 将同容量的空闲分区单独设立一个链表，再在内存中设立一张管理索引表，每个表象对应一个大小的空闲分区的链表的表头指针
   2) 分配时一般不进行分割（直接把碎片分给进程）
   3) 查找效率高，一般考虑到了大空间（只是很浪费而已）
2. 伙伴系统(buddy system)
   1) 每个区的大小都是2的k次幂，同大小的空闲分区单独设立一个双向链表
      (1) 先计算恰大于等于请求的2的i次幂，并尝试找到i的双向链表
      (2) 若找不到i，则不断尝试找i++，直至找到I
      (3) 不断将I割成两个I-1伙伴块，其中一个插入I-1的表，直至割到i

(4) 回收时可能要跟伙伴块不断合并，直至没有伙伴块

2) 伙伴块地址$buddy_k(x)=x+2^k$（x%2^(k+1)==0时）

(1) 或者 $buddy_k(x)=x-2^k$（x%2^(k+1)==2^k时）

3) 时间性能差于快速适应，高于顺序搜索

4) 空间性能优于快速适应，差于顺序搜索

3. 哈希算法/散列表算法

1) 构造空闲分区大小为关键字的哈希表，每个表项记录了对应链表头指针

2) 通过哈希函数，由空闲分区大小可快速找到对应空闲分区链表

六. 动态可重定位分区分配

1. 紧凑/拼接

1) 碎片/零头：不能被利用的小分区

2) 移动内存中的作业，使他们相邻接，碎片拼接成一个大分区

3) 每次紧凑完都要对移动过的程序/数据重定位

4)

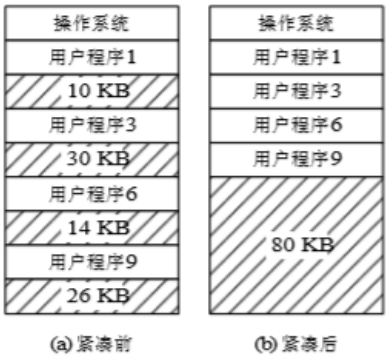


图 4-9 紧凑的示意

2. 动态重定位：地址变换过程在程序运行期间，随访问指令/数据而进行

1) 在系统中增设一个重定位寄存器，存储程序/数据在内存中的起始地址

(1) 减小地址转换对指令执行速度的影响

2) 移动指令/数据时不用对程序做修改了，只需更新内存起址

3)

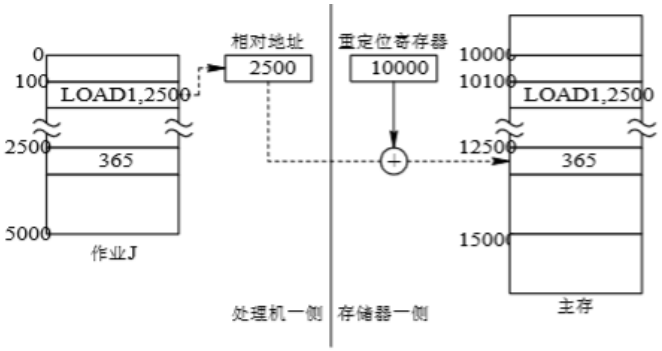


图 4-10 动态重定位示意图

3. 动态重定位分区分配算法

1) 该算法在找不到足够大的空闲分区时，若发现碎片和也小于用户要求，才返回分配失败信息；否则进行紧凑，并将拼接出的空间分配给它

2)

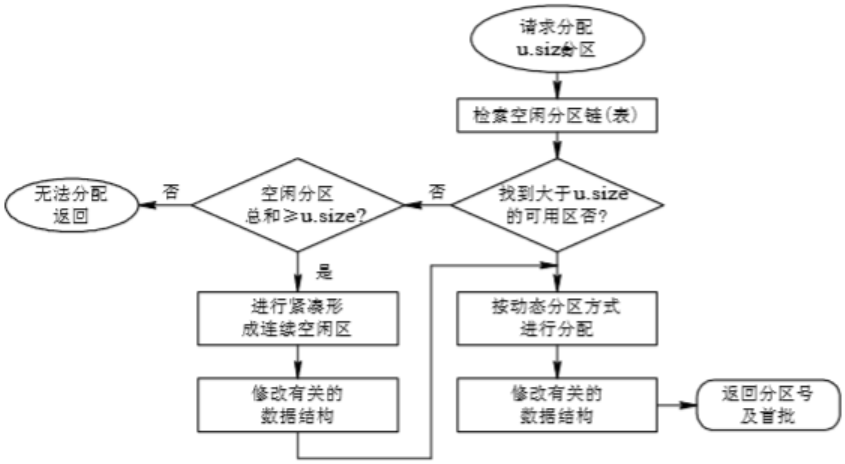


图 4-11　动态分区分配算法流程图

i.
ii.
iii.
iv.
v.
vi.
vii.
viii.
ix. ---------我是底线--------